Mitigating Tail Response Time of n-Tier Applications: The Impact of Asynchronous Invocations

QINGYANG WANG and SHUNGENG ZHANG, Louisiana State University–Baton Rouge, USA
YASUHIKO KANEMASA, Fujitsu Laboratories Ltd., Japan
CALTON PU, Georgia Institute of Technology, USA

Consistent low response time is essential for e-commerce due to intense competitive pressure. However, practitioners of web applications have often encountered the long-tail response time problem in cloud data centers as the system utilization reaches moderate levels (e.g., 50%). Our fine-grained measurements of an open source n-tier benchmark application (RUBBoS) show such long response times are often caused by Cross-tier Queue Overflow (CTQO). Our experiments reveal the CTQO is primarily created by the synchronous nature of RPC-style call/response inter-tier communications, which create strong inter-tier dependencies due to the request processing chain of classic n-tier applications composed of synchronous RPC/thread-based servers. We remove gradually the dependencies in n-tier applications by replacing the classic synchronous servers (e.g., Apache, Tomcat, and MySQL) with their corresponding event-driven asynchronous version (e.g., Nginx, XTomcat, and XMySQL) one-by-one. Our measurements with two application scenarios (virtual machine co-location and background monitoring interference) show that replacing a subset of asynchronous servers will shift the CTQO, without significant improvements in long-tail response time. Only when all the servers become asynchronous the CTQO is resolved. In synchronous n-tier applications, long-tail response times resulting from CTQO arise at utilization as low as 43%. On the other hand, the completely asynchronous n-tier system can disrupt CTQO and remove the long tail latency at utilization as high as 83%.

CCS Concepts: • **General and reference → Performance**; *Measurement*; *Experimentation*; **Design**; • **Information systems → E-commerce infrastructure**;

Additional Key Words and Phrases: n-tier systems, asynchronous, performance, scalability, cloud computing

ACM Reference format:
Qingyang Wang, Shungeng Zhang, Yasuhiko Kanemasa, and Calton Pu. 2019. Mitigating Tail Response Time of n-Tier Applications: The Impact of Asynchronous Invocations. *ACM Trans. Internet Technol.* 19, 3, Article 36 (July 2019), 25 pages.
https://doi.org/10.1145/3340462

This research has been partially funded by National Science Foundation by CISE's CNS (Grants No. 1566443 and No. 1421561), SAVI/RCN (Grants No. 1402266 and No. 1550379), CRISP (Grant No. 1541074), SaTC (Grant No. 1564097) programs, an REU supplement (Grant No. 1545173), Louisiana Board of Regents under Grant No. LEQSF (2015-18)-RD-A-11, and gifts, grants, or contracts from Fujitsu, HP, Intel, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

Authors' addresses: Q. Wang and S. Zhang, School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, 3325 Patrick F. Taylor Hall, LA 70803, USA; emails: {qwang26, szhan45}@lsu.edu; Y. Kanemasa, Fujitsu Laboratories LTD., 1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-8588, Japan; email: kanemasa@jp.fujitsu.com; C. Pu, College of Computing, Georgia Institute of Technology, 266 Ferst Dr, Atlanta, GA 30332-0765, USA; email: calton@cc.gatech.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.

1533-5399/2019/07-ART36 $15.00
https://doi.org/10.1145/3340462

1 INTRODUCTION

Long-tail response time problem occurs when a large portion of normal requests finishing within milliseconds co-exists with a small percentage of requests with very long response time (VLRT). Long-tail response time is a big concern for e-commerce: an Amazon study [25] reported that every increase of 100ms in page loading time is positively correlated with 1% decrease in sales. Nevertheless, long-tail response time is a persistent challenge: practitioners in recent years continuously report their real-world problems [12, 13, 24, 27, 29, 61], despite its long history. Long-tail response time is a non-trivial puzzle: VLRT requests usually start to appear at moderate average system utilization (e.g., 50%). Long-tail response time can also be an elusive target: Executing the VLRT requests by themselves would only take milliseconds.

In this article, we study an important class of long-tail response time problems in n-tier systems, specifically, VLRT requests caused by dropped packets because of complex cross-tier interactions among classic servers that communicate through Remote Procedure Calls (RPC). Our focus on distributed system phenomena complements previous research on single servers [14, 41, 51, 52, 59]. We further restrict our focus to VLRT requests at moderate utilization levels, which distinguishes our study from long-tail response time due to skewed workloads [22]. Our study supports Mogul's argument [36] that the performance of a distributed system can be much more complicated than the behavior of a single server because of the complex dependencies among components.

A fine-grained timeline analysis shows that the following sequence of events occur when VLRT requests appear due to dropped packets at moderate average system utilization. (1) The episode of resource millibottlenecks, which lasts for a very short lifespan, for example, CPU saturated for tens to hundreds of milliseconds due to transient events such as interference of co-located VMs. (2) Millibottleneck slows down or stops the server processing temporarily, triggering a process called Cross-tier Queue Overflow (CTQO) up and/or down the n-tier pipeline. (3) When a server's waiting requests grow beyond its maximum system queue capacity (e.g., worker thread pool is exhausted and the TCP buffer overflows), further incoming packets are dropped. (4) VLRT requests appear, because the dropped packets take several seconds to retransmit. This relatively long sequence of dependencies and events reveal the non-trivial nature of this class of VLRT requests. Furthermore, the VLRT requests are no longer so elusive, since now they can be reliably reproduced and analyzed by our fine-grained timeline analysis.

The first contribution of this article is the characterization of the VLRT requests caused by CTQO, which is a significant distributed system phenomenon with a large performance impact. Specifically, CTQO consists of two non-trivial components. The first component is the millibottlenecks that initiate CTQO. Our previous research [42, 56, 57] has reported millibottlenecks caused by Java garbage collection (Java GC), CPU dynamic voltage and frequency scaling (DVFS), and memory thrashing. These varied root causes of millibottlenecks make the solution to the problem more difficult and unlikely to be done systematically. In this article, we show two more case studies of millibottlenecks caused by interference of VM co-location and server log flushing. The second component is the cross-tier dependencies resulting from the synchronous RPC-style call/response communication between nodes. Such dependencies make a transient queuing effect to be propagated and amplified between different servers, leading to potential queue overflow and long response time requests. This makes us rethink the adoption of RPC-style synchronous invocations, despite its syntactic simplicity, in complex distributed systems such as e-commerce when the tail latency becomes a significant concern.

The second contribution is a systematic experimental evaluation of event-driven asynchronous servers in mitigating or preventing CTQO, originally created by the strong dependencies from the RPC-style servers in n-tier systems. Concretely, we replace the RPC-style synchronous servers in a three-tier web system with their asynchronous counterparts one by one. For example, the original thread-based Apache web server and Tomcat application server are replaced with the event-based Nginx and XTomcat, respectively. The thread-based MySQL is also replaced with a simulated event-based asynchronous MySQL by turning on a lightweight queue feature supported by the MySQL InnoDB storage engine. Our experimental results show that replacing either an upstream or a downstream synchronous server can only partially remove the CTQO problem. On the other hand, at the moderate to high utilization levels, the CTQO problem and the associated VLRT requests are effectively resolved only if all the thread-based servers are replaced with their asynchronous version.

The rest of the article is organized as follows. Section 2 shows the class of long-tail response time problems resulting from dropped packets. Section 3 experimentally illustrates the sequence of causal events that start from millibottlenecks and end in dropped packets due to CTQO. Section 4 describes the methodical evaluation of a three-tier system by replacing each component server with its asynchronous version. Section 5 summarizes the related work and Section 6 concludes the article.

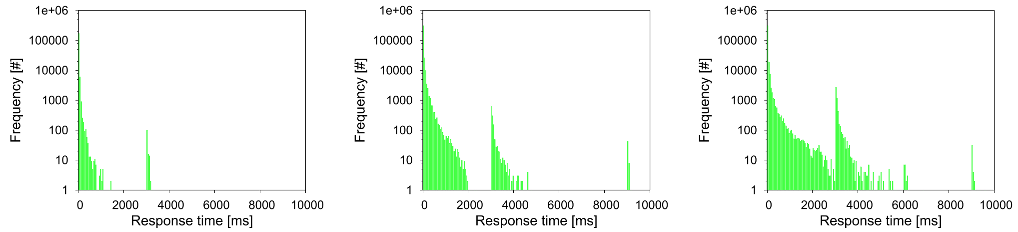
2 LONG-TAIL RESPONSE TIME DUE TO DROPPED PACKETS

The long-tail response time problem in n-tier web-facing applications has received increasing attention from practitioners and researchers in recent years [12, 24, 29, 61]. It is an interesting and challenging problem, because it consists of the co-existence of a majority of very fast responses (order of milliseconds) with a small number (but non-negligible) of very long response time (VLRT) requests that typically lasts several seconds. There are several known causes of VLRT requests, including skewed work requirements and dropped packets. In the class of long-tail response time problems due to skewed work requirements [22], some requests are significantly heavier (e.g., requests with more complicated business logic) than others and the long-tail response time is caused by such heavy requests. Such a class of long-tail response time problems is outside the scope of this article, since it saturates a server (higher than moderate utilization) and requires the (re)allocation of additional resources. As a concrete example, web search queries have simple syntax and semantics, but the queries with popular terms have several orders of magnitude more matches; without significant additional resources, such queries may become VLRT requests.

Instead of skewed work requirements, we focus on the class of VLRT requests resulting from dropped packets at moderate system utilization. Such class of VLRT requests are challenging and interesting because of two apparently contradictory factors. First, the VLRT requests in this class are not intrinsic long requests; they only take milliseconds to execute when run by themselves. So this class of VLRT requests are caused by either waiting or queuing somewhere in the system. Second, this class of VLRT requests appear when the system is at moderate average system utilization levels (e.g., 50%), so queuing is usually considered mild and would not cause VLRT requests based on the classic queuing models.

Our experimental results show that VLRT requests caused by dropped packets help form a long-tail multi-modal response time distribution. For example, Figure 1 shows the response time distribution of a three-tier benchmark web application at three different workload levels. The detailed experimental setup is in Appendix A. This figure shows that most requests finish within a few hundreds of milliseconds, however, a few clusters of long requests start at 3, 6, and 9s, illustrating the tail latency of the target benchmark application. Our previous results [56, 57] have shown that these long requests clusters at certain response times are due to the TCP retransmission mechanism for the dropped TCP packets (the minimum TCP retransmission time-out is 1s [43]).

Figure 1 also shows that VLRT requests appear at the system resource utilization as low as 43%. So it is clear that those VLRT requests are not caused by persistent resource bottlenecks. As the average system utilization continues to increase, the VLRT requests occur more frequently as shown



(a) 4,000 workload case. The system throughput is 572 req/s. The highest average CPU util. among servers is 43%.

(b) 7,000 workload case. The system throughput is 990 req/s. The highest average CPU util. among servers is 75%.

(c) 8,000 workload case. The system throughput is 1,103 req/s. The highest average CPU util. among servers is 85%.

Fig. 1. System response time distribution (semi-log graph) as system workload increases. The long-tail response time problem appears far before the system saturation.

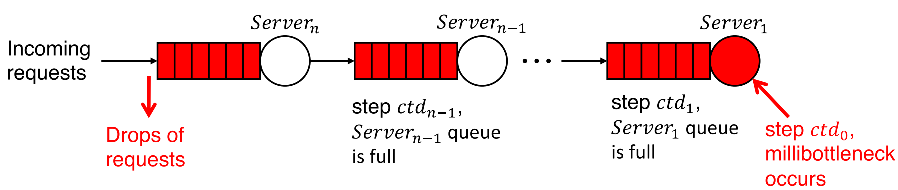


Fig. 2. Illustration of cross-tier dependency model under millibottlenecks. A millibottleneck occurs (step ctd_0) in $Server_1$ and makes $Server_1$ queue full (step ctd_1), results in queue overflow towards upstream until $Server_n$ (step ctd_n), eventually leading to drops of requests in $Server_n$ and VLRT requests.

in Figures 1(b) and 1(c). In fact, the percentage of VLRT requests over the total number of finished requests already exceeds 5%, which is considered as a severe violation of the Service Level Agreements (SLAs) by most e-commerce websites [19, 25, 31]. In the next section, we will study two cases illustrating that *millibottlenecks* are linked to the dropped packets and the resulting VLRT requests. Concretely, a millibottleneck will initiate *Cross-tier Queue Overflow (CTQO)*, a sequence of causal events that will lead to VLRT requests because of dropped packets and TCP retransmissions.

3 CROSS-TIER QUEUE OVERFLOW BY MILLIBOTTLENECKS

3.1 Cross-tier Dependency Model

We start our discussion by defining a cross-tier dependency model that starts from millibottlenecks, but independent of specific causes of millibottlenecks. This is important, because several very distinct causes of millibottlenecks have been found, including system software (e.g., JVM garbage collector [47], at architecture level (e.g., Dynamic Voltage and Frequency Scaling in anti-synchrony with a bursty workload [56]), and two other classes of millibottlenecks described in the following sections: CPU millibottlenecks due to VM consolidation, and I/O millibottlenecks due to log flushing. Despite the variety of the causes, the sequence of events that follow millibottlenecks is the same. We call such a sequence a *Cross-tier Dependency Sequence*, because its components are tied together by strong dependencies between the synchronous servers due to their RPC-style request-response communications.

A Cross-tier Dependency Sequence (denoted by steps $ctd_0, \ldots, ctd_n$ in an n-tier system) starts from a millibottleneck over some concrete resource (e.g., CPU or I/O) in Server_1, as shown in Figure 2. (The millibottleneck is called step ctd_0 in recognition of its originating role, but it continues through the entire sequence, overlapping with the remaining steps ctd_1 through ctd_n.) The resource saturation in ctd_0 causes Server_1's threads to queue up (step ctd_1), waiting for the

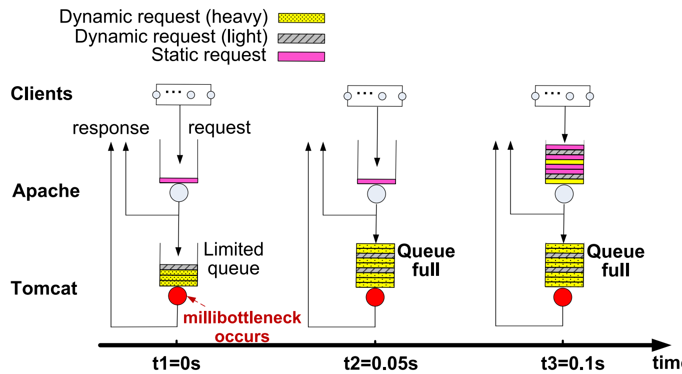


Fig. 3. A simple upstream CTQO illustration between Apache and Tomcat. A millibottleneck occurs in Tomcat at t_1. Tomcat queue fills up at t_2. Then all types of requests are pushed back to queue in the upstream Apache at t_3, causing queue amplification and overflow in Apache.

bottlenecked resource. In typical web-facing applications, the queues that store the waiting requests consist of Server_1's thread pool (size determined by its configuration parameter, e.g., 150) and TCP buffer (default size of 128). We denoted *MaxSysQDepth* as the total number of requests that can be queued in all those queues. During a millibottleneck, quick arrival of jobs (typically on the order of several thousand per second in web-facing applications with a response time of a few milliseconds) can exceed MaxSysQDepth(Server_1). This "filling up" of all the local request-handling queues up to MaxSysQDepth(Server_1) forms the step ctd_1 of Cross-tier Dependency Sequence.

Step ctd_1 ends when the number of queued requests exceeds MaxSysQDepth(Server_1) and Server_1 becomes unable to accept new requests from the upstream Server_2. In a classic RPC-style implementation, Server_2 blocks and occupies a thread for each pending request. The "filling up" of the queues in Server_2 forms the step ctd_2. As the millibottleneck progresses, Server_2 fills up its queues (ctd_2) and the next-in-line upstream Server_3 starts to see its threads becoming blocked (step ctd_3). The process continues upstream (to step ctd_n) until one of the Server_i (where $1 \leq i \leq n$, but often $i = n$) starts to drop packets, resulting in VLRT requests. We note tens of thousands or even more requests may be processed per second in a large size of system. Our model still applies to such large systems as long as each server adopts synchronous/blocking RPC for inter-server communication. Concretely, once a server experiences a millibottleneck, requests are pushed back to queue in the upstream servers along the chain of dependencies, causing queue amplification and overflow.

This process of a growing Cross-tier Dependency Sequence toward upstream, eventually leading to VLRT requests is called *upstream CTQO* (Cross-tier Queue Overflow). Figure 5(b) shows that the Apache queues (Server_2 in ctd_2) grow much longer than those in Tomcat (Server_1 in ctd_1). This is due to upstream CTQO shown in Figure 3. At time t_1, Tomcat millibottleneck starts (ctd_0). At t_2, hundreds of requests arrive at Tomcat (ctd_1) and reach MaxSysQDepth(Tomcat), starting the ctd_2 in Apache. At t_3, both the dynamic and static requests (e.g., images) start to queue up in Apache (ctd_2), which can grow significantly longer than Tomcat. This is shown in Figure 6(a), by the categorization of queued requests in SysSteady-Apache during the same 20s timeframe as in Figure 5(b).

To illustrate the Cross-tier Dependence Sequence in upstream CTQO, our experiments use a normal three-tier configuration of RUBBoS [40], a representative n-tier application benchmark modeled after Slashdot. The RUBBoS workload is a set of emulated clients sending various HTTP requests (both static and dynamic) generated by a Markov-chain user behavior model for web-facing

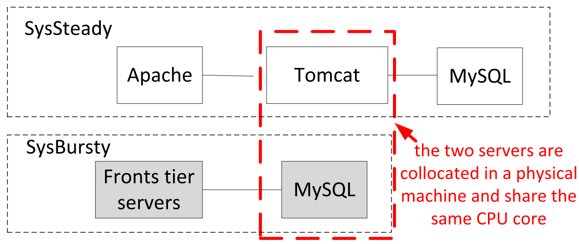


Fig. 4. Illustration of VM Consolidation setup: SysSteady Tomcat shares the same CPU core with SysBursty MySQL.

e-commerce applications. The smallest system configuration consists of one Apache HTTP server, one Tomcat Application Server, and one MySQL database server, with more details in Appendix A.

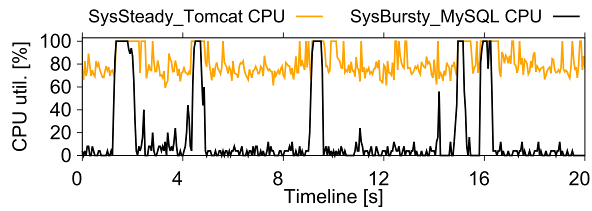
The first illustrative example of milli-dependency sequences is created by a CPU millibottleneck created by interferences among consolidated VMs. Sharing infrastructure resources through VM consolidation is a common practice for cloud computing platforms to reduce operational cost and gain high return on investment (RoI) [5, 17, 23, 33]. A typical profitable scenario of consolidation is to co-locate multiple under-utilized VMs on the same physical machine by following certain rules such as the classic bin-packing algorithm [21]. However, there is potential interference among VMs when high CPU demand from multiple VMs coincide [37], e.g., in naturally bursty [34] web-facing applications.

We co-locate two RUBBoS three-tier applications, named SysSteady and SysBursty (Figure 4), in our VM consolidation experiments. For clarity of analysis, there is only one shared node, with the SysSteady Tomcat co-located with SysBursty MySQL and they share the same CPU core. Other servers of each system run on dedicated physical machines. SysSteady serves 7,000 normal RUBBoS client users while SysBursty serves only 400 client users but with a burst index 100 times higher (see Reference [35]) than SysSteady. Such a bursty workload is common in web-facing applications (sometimes referred to as the "Slashdot" effect [1]).

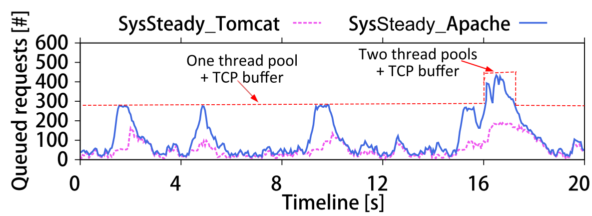
3.2 CPU Millibottleneck Caused by VM Co-location

Figure 5(a) shows episodes of CPU millibottlenecks (ctd_0) due to the interference between the two co-located VMs at 2, 5, 7–9, and 12s. These millibottlenecks cause queuing in SysSteady Tomcat (ctd_1), which leads to even longer queue in Apache (ctd_2) due to Cross-tier Queue Amplification, as shown in Figure 5(b). The Cross-tier Queue Amplification happens, because every queued request in Tomcat consumes one thread in Tomcat and one connection between Apache and Tomcat. We note that queuing in Apache happens purely due to the waiting for Tomcat, since none of Apache resources is saturated.

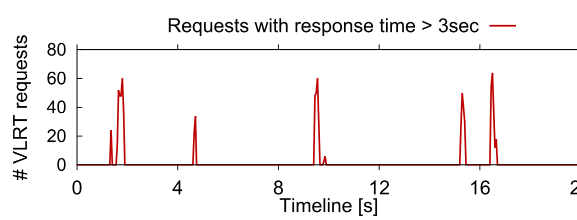
For millibottlenecks of moderate length, the growth of Apache queues (ctd_2) leads to queue overflow (the dashed line in Figure 5(b) indicates Apache runtime queue limit) and dropped packets. These dropped TCP packets, being retransmitted after a certain amount of time (3s for the first retransmission in Redhat kernel 2.6.32), lead to the VLRT requests as shown in Figure 5(c). Figure 5(b) shows two levels of queue overflow. The first level queue overflow occurs at 2, 5, 10, 15s, because the queued requests reach to the limit 278 (sum of thread pool size 150 and TCP buffer size 128). Once queued requests in Apache exceed such a limit, new incoming requests will be dropped and retransmitted, resulting in long requests. The second-level queue overflow occurs at about 17s. This happened when all the worker threads in the first process are occupied, and during the creation of a second Apache process with an additional thread pool of size 150. However, new incoming requests still get dropped when the second Apache process is in the initiation period.



(a) Bursty CPU utilization of SysBursty-MySQL make SysSteady-Tomcat also perceive 100% of CPU at time markers 2, 5, 9, and 15s since the CPU core is shared.



(b) SysSteady Tomcat queue fills up during the millibottleneck periods, causing queue amplification and overflow in SysSteady Apache.



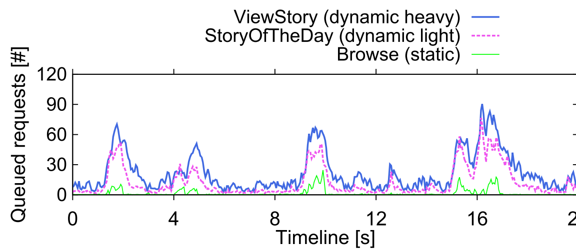
(c) VLRT requests (>3s) appear during the millibottleneck periods. Such VLRT requests are due to queue overflow in Apache and TCP retransmissions.

Fig. 5. Illustration of millibottlenecks in Tomcat causing upstream CTQO in the VM consolidation experiments.

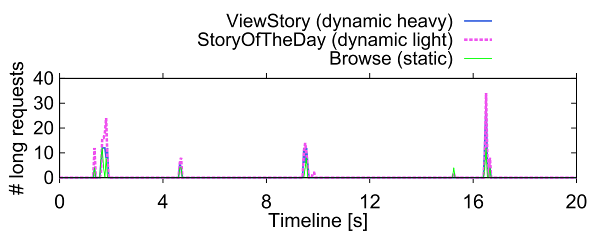
This is because initiating a new process with a large size thread pool consumes non-trivial CPU resources and blocks Apache for a short period of time (tens of milliseconds).

Figure 5(b) shows that the Apache queues (ctd_2) grow much longer than those in Tomcat (ctd_1). This occurs because of the Cross-tier Queue Amplification as illustrated in Figure 3. At t_1 time marker, Tomcat millibottleneck starts (ctd_0). At t_2, hundreds or even thousands of requests rush to Tomcat and fill up the Tomcat queue (ctd_1), blocking incoming requests. At t_3, Cross-tier Queue Amplification then causes all types of requests to queue in the upstream Apache (ctd_2), including both the dynamic and static requests. Dynamic requests, regardless of being light and heavy,[1] are indistinguishably queued in FIFO order in Apache, causing longer queues. Since it is common that a web server serves more static/local requests (e.g., images, HTML, CSS) than dynamic requests, the queueing effect in Apache can be significantly amplified. This is shown in Figure 6(a), by the breakdowns of queued requests in SysSteady-Apache during the same 20s timeframe as in Figure 5(b). All types of requests, including the static "Browse" requests, are indeed queued in Apache during ctd_2.

[1]A dynamic request being heavier than a light one can be attributed to a heavy request consuming significantly more system resources than the light one.



(a) Breakdowns of queued requests in *SysSteady-Apache* (three representative types out of eight are shown here) during the same 20s timeframe as in Figure 5(b). This figure shows all types of requests, including the static "Browse" requests, are queued in Apache, though the millibottlenecks occur in Tomcat.



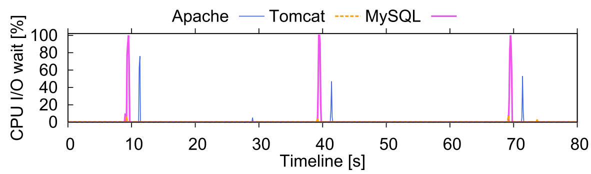
(b) Breakdowns of VLRT requests in *SysSteady* (Figure 5(c) shows the aggregation). This figure shows that VLRT requests can be any type.

Fig. 6. Categorization of queued requests in *SysSteady-Apache* during the same timeframe of Figure 5(b). Three (out of eight) representative request types show that both static and dynamic requests are queued in Apache during a millibottleneck in Tomcat.

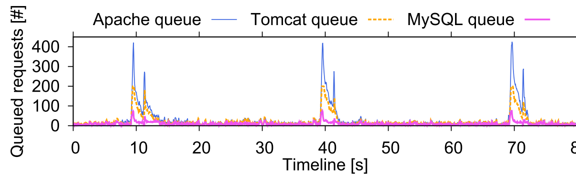
3.3 I/O Millibottleneck Caused by Log Flushing

The second example illustrating the milli-dependency sequences is background monitoring activities common in large scale distributed systems. Our experiments use one Apache server, one Tomcat, and one MySQL (see Figure 15(c)). We add three more CPU cores to the original Tomcat VM (now totally four cores) to avoid the CPU bottleneck in Tomcat. This upgrade of Tomcat allows millibottlenecks to appear in MySQL. Specifically, the monitoring tool `collectl` [45] monitors (at fine granularity—every 50ms) the utilization of system sources such as CPU, memory, network, disk I/O, and process runtime state in each server. `collectl` has a control knob for users to specify how frequent to flush the accumulated measured data from memory to disk (no dedicated core for disk I/O activities). We set the time interval to 30s, a common choice that reduces the monitoring interference with the running application.

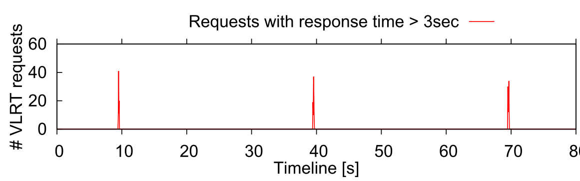
Figure 7(a) shows millibottlenecks in MySQL (ctd_0) in red high peaks at 10, 40, and 70s (30s intervals). These are due to `collectl` flushing monitoring data from memory to disk, with I/O wait for MySQL reaching 100%. These transient CPU I/O waits create millibottlenecks that lead to other threads blocking for CPU (ctd_1). When all of the MySQL threads become blocked, the upstream server (Tomcat) starts to block threads and see growing queues (ctd_2) due to Cross-tier Queue-amplification, shown in Figure 7(b). The queues in Tomcat, when long enough, cause further Cross-tier-queue Amplification to Apache (ctd_3). Once the queued requests in Apache consume all the queue slots (threads and TCP buffer), packets are dropped and VLRT requests are created by TCP retransmission as shown in Figure 7(c).



(a) CPU I/O wait millibottlenecks appear in MySQL because `collectl` flushes measurements every 30s



(b) Millibottlenecks in MySQL trigger upstream CTQO to Tomcat and Apache. Apache drops new coming requests when MaxSysQDepth(Apache) is exceeded



(c) VLRT requests appear during the millibottleneck periods. Such VLRT requests are due to queue overflow in Apache and TCP retransmissions.

Fig. 7. I/O millibottlenecks in MySQL causing upstream CTQO in the log flushing scenario.

4 EVALUATION OF ASYNCHRONOUS INVOCATION IN N-TIER SYSTEMS

4.1 Evaluation Method

The experiments in Section 3 show both the variety and the importance of Cross-tier Queue Overflow in the presence of inter-dependent nodes. Since the dependencies created by synchronous RPC-style request-response are a necessity for upstream CTQO, we proceed to remove these dependencies by replacing the RPC request-response with asynchronous invocation and evaluate the interactions between CTQO and VLRT requests.

We acknowledge that since Birrell and Nelson's classic 1984 paper [7], RPC has been widely used to build distributed systems such as n-tier applications. However, our experimental evidence shows that system designers and implementers should consider a return to asynchronous communications that preceded RPC when upstream CTQO and long-tail response time become a real problem. The experiments also reveal some of the underlying complexities of the problem. For instance, replacing one server in the chain (e.g., Nginx instead of Apache) reduces but does not eliminate all the long-tail response time problems.

We will use the same three-tier application benchmark of Section 3 as a baseline, but replace each of the three thread-based RPC-style servers (Apache, Tomcat, and MySQL) one-by-one and evaluate the performance impact of asynchronous invocation instead of RPC. Figure 8 shows the three asynchronous servers (Nginx [38], XTomcat [4, 16], and XMySQL) in the final configuration. More details regarding the asynchronous servers and benchmark application implementation can

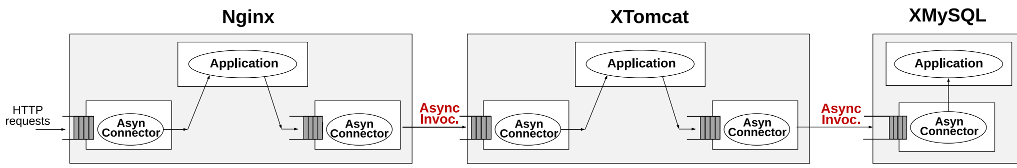


Fig. 8. Illustration of the asynchronous three-tier system architecture. We use Nginx, XTomcat, and XMySQL to replace the original thread-based RPC-style Apache, Tomcat, and MySQL, respectively.

be found in the Appendices B and C. For clarity of presentation, we use the term NX to denote the number of asynchronous servers in a set of experiments.

4.2 NX=1, Replacing Apache with Nginx

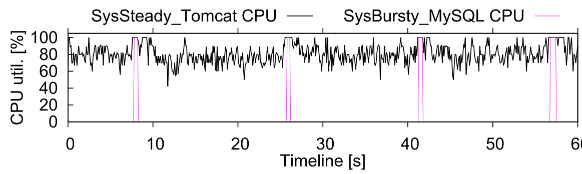
Our study in Sections 3.2 and 3.3 shows that Apache causes VLRT requests by dropping packets because of upstream CTQO, thus, a natural hypothesis is that we may solve the upstream CTQO problem by replacing just the thread-based RPC-style Apache with an asynchronous web server. The answer turns out to be partially true. The asynchronous event-based web server such as Nginx [38] won't drop packets indeed, however, the problem moves to the downstream tiers; for example, the synchronous Tomcat and MySQL start to drop packets and the long-tail response time problems arise again.

We make two changes of the experimental setup compared to that in the previous Section 3. First, to better control the occurrence of millibottlenecks in our VM consolidation experiments (Section 3.2), we modified SysBursty workload generator to launch specific request bursts at fixed intervals. For example, the workload generator launches a burst of 400 ViewStory requests at 15s intervals, which will trigger CPU millibottlenecks with an approximately 300ms length. Second, we replace the thread-based RPC-style Apache with the asynchronous Nginx to effectively remove the queue limit of MaxSysQDepth(Apache). The concurrent request processing in Nginx is no longer limited by its thread pool size, but a lightweight queue with size LiteQDepth in Nginx, where LiteQDepth $\gg$ MaxSysQDepth(Apache). This change indeed enables Nginx to route all the requests to the downstream Tomcat, thus shifting the problem downstream. Under the assumption that the web server is not the bottleneck (which is the case of the RUBBoS benchmark), the downstream Tomcat and MySQL potentially encounter millibottlenecks at a moderate utilization level.

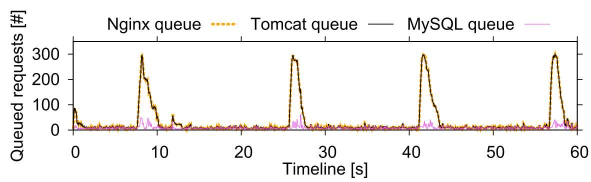
The first problem that can arise is due to millibottlenecks in Tomcat. Figure 9 shows the experimental results of millibottlenecks in SysSteady Tomcat (co-located with the MySQL VM of SysBursty as in Figure 4). Figure 9(a) shows the SysSteady Tomcat and MySQL CPU utilization (we omit the CPU utilization of Nginx, since it is always less than 40%). We can see several millibottlenecks in SysSteady Tomcat (at time mark 7, 26, 42, 57, and 72s), due to the co-located MySQL VM of SysBursty processing a burst of requests.[2] The millibottlenecks in Tomcat cause Tomcat queue to grow as shown in Figure 9(b). Since Nginx is asynchronous, Nginx will route all the requests it receives (up to LiteQDepth(Nginx)) to the downstream Tomcat, which can hold concurrent requests up to 293 (sum of Tomcat thread pool size 165 and TCP buffer size 128 by default). As the new coming requests from Nginx overwhelm the queues in downstream Tomcat (in a process we call *downstream CTQO*) packets are dropped, and become VLRT requests as shown in Figure 9(c).

The second problem that can arise is originated from millibottlenecks in MySQL. Figure 10 shows the millibottlenecks in MySQL VM of SysSteady, caused by bursts from the co-located

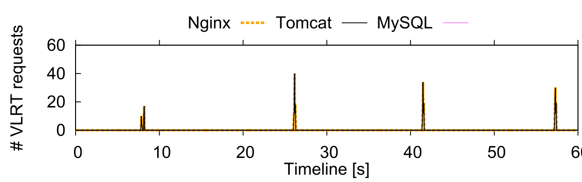
[2]The sudden drop of MySQL CPU in SysBursty is due to Tomcat *not* sending requests downstream because of the millibottleneck.



(a) Bursts CPU utilization of SysBursty-MySQL make the co-located SysSteady-Tomcat perceives transient CPU saturation at time markers 7, 26, 42, and 57s.



(b) SysSteady Tomcat queue reaches MaxSysQDepth(Tomcat)=165+128=293 during the millibottleneck periods. Nginx queue overlaps with Tomcat queue.

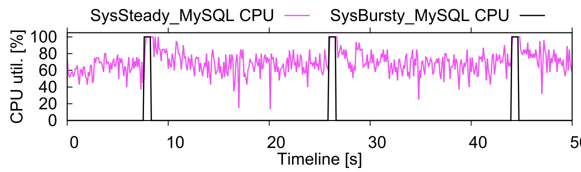


(c) VLRT requests appear during the millibottleneck periods. Such VLRT requests are due to queue overflow in SysSteady Tomcat and TCP retransmissions.

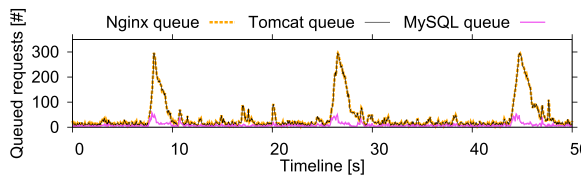
Fig. 9. NX=1, Nginx-Tomcat-MySQL configuration when millibottlenecks occur in Tomcat. No upstream CTQO observed in Nginx, but queue overflow happens in Tomcat during the millibottlenecks.

MySQL VM of SysBursty. Figure 10(a) shows the SysSteady MySQL CPU utilization, with millibottlenecks (ctd_0) at time marks 8, 26, and 45s. These millibottlenecks cause MySQL to queue (Figure 10(b)); the queue size of which is up to 50 (equal to DB connection pool size in Tomcat). When the queued requests exceed MaxSysQDepth(MySQL) in step ctd_1, the synchronous Tomcat starts to queue the incoming requests (ctd_2) due to the upstream CTQO between Tomcat and MySQL. When the arriving requests (up to LiteQDepth(Nginx)) exceed queue limit of Tomcat, excess requests will be dropped, leading to VLRT requests because of TCP retransmission (Figure 10(c)). This process between MySQL and Tomcat forms upstream CTQO, a similar process to the one we introduced between Apache and Tomcat (see Figure 5(c)).

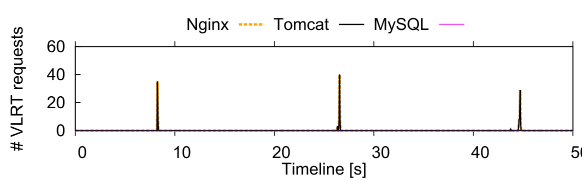
These experiments show that replacing the thread-based RPC-style Apache with the event-driven asynchronous Nginx removed the front-most web server tier from the chain of Cross-tier dependencies. However, new problems arise downstream. First, when Tomcat encounters millibottlenecks, the asynchronous Nginx is able to route excess requests (up to LiteQDepth(Nginx)) to the downstream Tomcat, causing downstream CTQO, since LiteQDepth(Nginx) $\gg$ MaxSysQDepth(Tomcat). The excess requests (LiteQDepth(Nginx) – MaxSysQDepth(Tomcat)) are dropped, becoming VLRT requests due to TCP retransmission. Second, when MySQL encounters millibottlenecks, they may cause upstream CTQO between MySQL and Tomcat, in a way analogous to the millibottlenecks in Tomcat causing upstream CTQO in Apache (Sections 3.2 and 3.3).



(a) Bursts CPU utilization of SysBursty-MySQL make the co-located SysSteady-MySQL perceives transient CPU saturation at time markers 8, 26, and 45s.



(b) Millibottlenecks in SysSteady MySQL make requests to queue in MySQL, but much longer queues in Tomcat and Nginx, suggesting an upstream CTQO during the millibottleneck periods.



(c) VLRT requests observed in SysSteady Tomcat. Such VLRT requests are due to CTQO in SysSteady Tomcat and TCP retransmissions.

Fig. 10. NX=1, Nginx-Tomcat-MySQL configuration when millibottlenecks occur in MySQL. Upstream CTQO observed between MySQL and Tomcat during the millibottleneck periods.

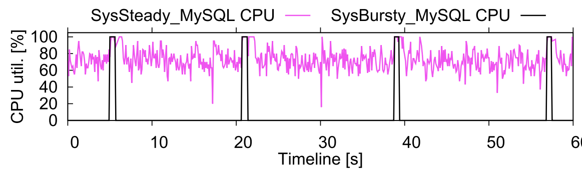
4.3 NX=2, Replacing Tomcat with XTomcat

After we replace Apache with Nginx in the previous section, the following step is to replace the thread-based RPC-style Tomcat with an event-based asynchronous application server. Without a popular asynchronous application server, we transform the original thread-based Tomcat to its asynchronous version called XTomcat. Our focus here is to evaluate the impact of XTomcat on CTQO, so we put the transformation of Tomcat into XTomcat in Appendix B.

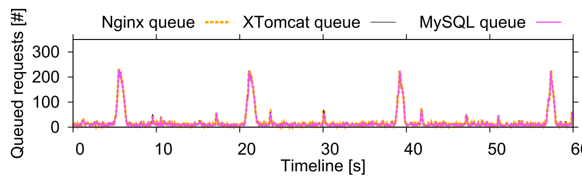
We want to know whether using both asynchronous Nginx and XTomcat in a simple three-tier system will address the CTQO problem and avoid VLRT requests. The answer turns out to be partially true again. Our experimental results show the upstream and downstream CTQO between Nginx and XTomcat are indeed fixed. However, downstream CTQO continues to appear in MySQL.

The first case of downstream CTQO appears when MySQL encounters millibottlenecks, created by co-locating MySQL VM of SysSteady with the MySQL VM of SysBursty. Figure 11(a) shows the SysSteady MySQL and SysBurst MySQL CPU utilization during a 60s time period. The millibottlenecks in SysSteady MySQL appear at time mark 5, 22, 38, and 56s, and make MySQL to queue (Figure 11(b)). Since MySQL queue limit is 228 (sum of thread pool size 100 and TCP buffer size 128), the excess requests from Nginx and XTomcat to the downstream MySQL could cause requests to drop, generating VLRT requests as shown in Figure 11(c).

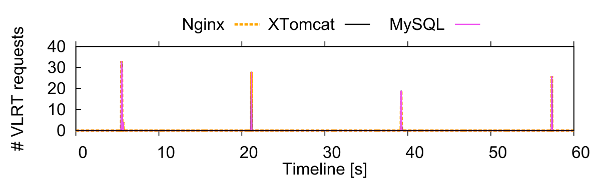
The second case of downstream CTQO appears when XTomcat encounters millibottlenecks, created by co-locating XTomcat VM of SysSteady with MySQL VM of SysBursty (Figure 12). The



(a) Bursts CPU usage of SysBursty-MySQL make the co-located SysSteady-MySQL perceives transient CPU saturation at time markers 6, 21, 39, and 57s.



(b) Millibottlenecks in SysSteady MySQL make requests to queue in MySQL up to MaxSysQDepth(MySQL), leading to dropped packets and VLRT requests (see panel (c)).



(c) VLRT requests observed in SysSteady MySQL. Such VLRT requests are due to queue overflow in MySQL and TCP retransmissions.

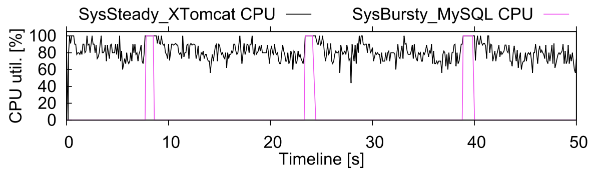
Fig. 11. NX=2, Nginx-XTomcat-MySQL configuration when millibottlenecks occur in MySQL. No upstream CTQO observed in XTomcat and Nginx. However, downstream CTQO observed when queued requests in MySQL exceeds MaxSysQDepth (MySQL).

CPU utilization of XTomcat and MySQL (Nginx omitted due to low utilization) are shown in Figure 12(a). We can see millibottlenecks in XTomcat make XTomcat to queue at time marks 8, 24, and 39s, shown in Figure 12(b). When a millibottleneck in XTomcat ends, all the queued requests in XTomcat are quickly routed to the downstream MySQL (Figure 12(b)). Since XTomcat is able to store up to LiteQDepth(XTomcat) requests, if LiteQDepth(XTomcat) > MaxSysQDepth(MySQL), then we have downstream CTQO with the excess packets dropped, creating VLRT requests.

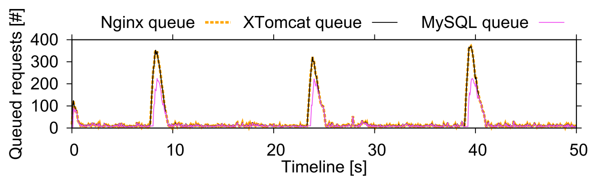
The downstream CTQO between XTomcat and MySQL can happen in realistic workloads. Suppose XTomcat has a millibottleneck that lasts 0.4s, and the average request arrival rate for the application is 1,000 req/s. XTomcat will store 400 requests during the millibottleneck, since LiteQDepth(XTomcat) is large (e.g., 65,535 occupied TCP port numbers). In this case, 400 requests will be quickly pushed to MySQL, causing downstream CTQO and dropped requests when the number of inflowing requests is larger than MaxSysQDepth(MySQL), which is 228 (Figure 12(c)).

4.4 NX=3, Replacing MySQL with XMySQL

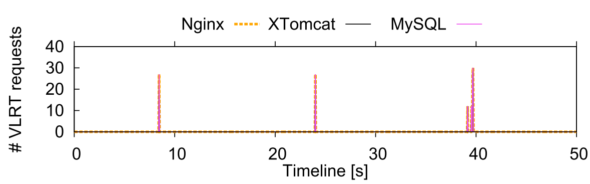
We further evaluate CTQO after we replace the last synchronous server in the system—MySQL with XMySQL, an asynchronous event-based database server. We want to know whether replacing all the component servers in the system with their asynchronous versions (e.g., Nginx, XTomcat, and XMySQL) will address CTQO and avoid VLRT requests. Our experimental results show that



(a) Bursts CPU usage of SysBursty-MySQL make the co-located SysSteady-XTomcat perceives transient CPU saturation at time markers 8, 24, and 39s.



(b) Millibottlenecks in SysSteady XTomcat make requests to queue in XTomcat. However, the queued requests in Tomcat are sent to MySQL in a batch when each millibottleneck period ends, causing downstream CTQO in MySQL.



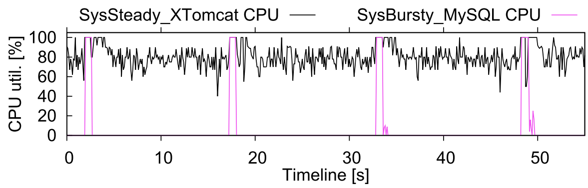
(c) VLRT requests observed in SysSteady MySQL. Such VLRT requests are due to downstream CTQO from Tomcat to MySQL, causing dropped packets.

Fig. 12. NX=2, Nginx-XTomcat-MySQL configuration when millibottlenecks occur in XTomcat. Downstream CTQO observed when batched requests are flushed from XTomcat to MySQL, causing queue overflow in MySQL and dropped packets.

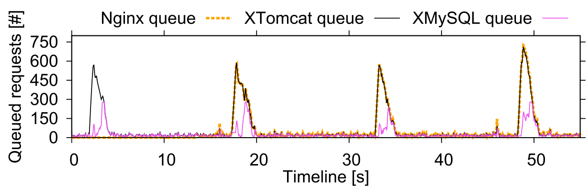
the CTQO (both upstream and downstream) can be prevented once all the servers in the three-tier system become asynchronous, thus avoid VLRT requests regardless of millibottlenecks occurring in any server. XMySQL simulates an asynchronous MySQL by adopting the InnoDB storage engine of MySQL, which supports a lightweight queue to store the waiting queries when the thread pool is fully utilized. Concretely, we configured 8 threads in the InnoDB storage engine to process active queries, and an additional lightweight queue with a size of 2,000 for waiting queries. Such a configuration is large enough for LiteQDepth(XMySQL).

We evaluate the asynchronous three-tier system using the same CPU millibottlenecks scenarios as we described in the VM co-location experiments in Section 3.2 and Figure 4. The thread-based RPC-style component servers (Apache-Tomcat-MySQL) in the original system SysSteady are replaced with their asynchronous versions (Nginx-XTomcat-XMySQL). We also changed the RUBBoS benchmark application to adopt asynchronous invocation for inter-tier communication.

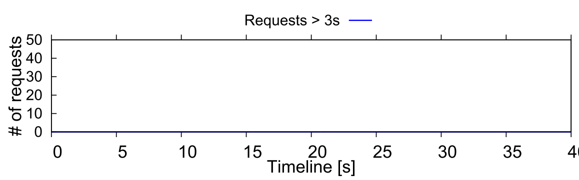
First, we evaluate the case when millibottlenecks occur in XTomcat. Figure 13(a) shows SysSteady XTomcat encountered CPU millibottlenecks at time marks 4, 13, and 35s, causing XTomcat to queue (Figure 13(b)). This figure also shows that the XTomcat queue and Nginx queue almost overlap with each other, suggestion no upstream CTQO between the two tiers. In addition, Nginx, XTomcat, and XMySQL have very large LiteQDepth (e.g., 65,535 and 2,000, respectively),



(a) Bursts CPU usage of SysBursty-MySQL make the co-located SysSteady-XTomcat perceives transient CPU saturation.



(b) Millibottlenecks in SysSteady XTomcat make requests to queue in XTomcat; Nginx and XTomcat queue overlap with each other, indicating to upstream CTQO observed in Nginx. Also no downstream CTQO observed in XMySQL since its LiteQDepth is large (e.g., 65,535).



(c) Number of VLRT requests counted at every 50ms tme window.

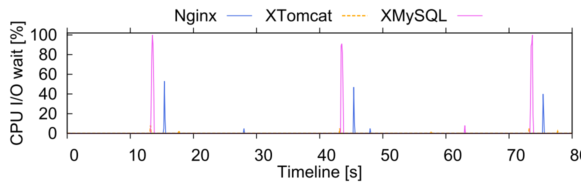
Fig. 13. NX=3, Nginx-XTomcat-XMySQL configuration when millibottlenecks occur in XTomcat. No upstream or downstream CTQO observed in the system.

thus avoiding downstream CTQO. Since there is no CTQO either upstream or downstream, we see no dropped requests, thus no VLRT requests.

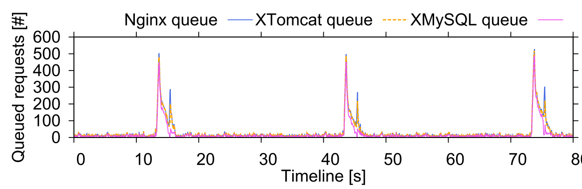
Second, we evaluate the case when millibottlenecks occur in XMySQL. We run the same Log Flushing experiments (I/O millibottlenecks) as described in Section 3.3. We only change the previous synchronous version of the three-tier system to its asynchronous counterpart. Figure 14(a) shows the CPU I/O wait of each tier. This figure shows XMySQL encounters I/O millibottlenecks at every 30s (time marks 13, 43, and 73s). Figure 14(b) shows the runtime queue of XMySQL, XTomcat and Nginx almost overlap, indicating no upstream CTQO between them. Also, large LiteQDepth(Nginx), LiteQDepth(XTomcat), and LiteQDepth(XMySQL) prevent the downstream CTQO, thus there are no dropped packets.

4.5 Discussion of Alternative Designs

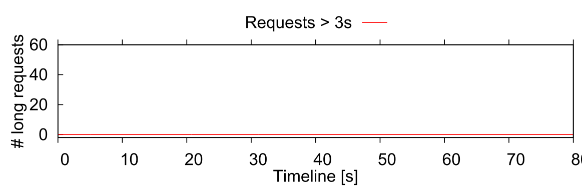
A straightforward fix for the thread-based RPC-style servers is to increase the MaxSysQDepth, for example, by increasing the worker thread pool size. This simple fix might mitigate or prevent the CTQO problems described in Section 3. However, increasing the thread pool size to thousands causes many other performance issues as discussed before [9, 26, 41, 58, 59]. Specifically, over-allocated threads cause overhead coming from various system layers such as LLC miss, high context switches, and scheduling overhead. Previous work [54, 59] has shown that significant multithreading overhead can be introduced even with tens to a few hundred threads, depending on the



(a) MySQL encounters millibottlenecks when `collectl` flushes log data every 30s.



(b) Queues of all the three servers overlap with each other, suggesting no CTQO in their lightweight queues during the millibottlenecks in MySQL



(c) No VLRT requests occur during the period of millibottlenecks in MySQL.

Fig. 14. NX=3, Nginx-XTomcat-XMySQL configuration when millibottlenecks occur in XMySQL. No upstream or downstream CTQO is observed in the system during the I/O millibottlenecks.

type of servers. In addition, over-allocated threads in Java-based servers also lead to non-linearly increased JVM garbage collection time resulting from high Memory footprint used for request processing [58].

Another straightforward fix is to increase the default TCP buffer size (128 in Linux kernel 2.6.32), the second component of MaxSysQDepth. However, it is also a non-trivial task to choose a reasonable size for TCP buffer size, since the workload for web application is very bursty by nature [8]. On the other hand, increasing network buffer sizes has been shown by the networking community to cause side effects such as bufferbloat [15], which causes long delivery latency.

For completeness, we also ran experiments on a configuration where a synchronous server is upstream from an asynchronous server (Apache-XTomcat-MySQL). The experiments confirm the intuitive reasoning that by itself the XTomcat is unable to prevent completely either the upstream CTQO or the downstream CTQO. First, when millibottlenecks occur in XTomcat, upstream CTQO causes Apache to drop packets in a situation similar to Apache-Tomcat outlined in Section 3.2. Second, after the millibottleneck in XTomcat ends, a burst of requests released by XTomcat causes downstream CTQO in MySQL, which is the same case described in Section 4.3. For carefully chosen configurations, we can observe the simultaneous occurrence of both upstream CTQO and downstream CTQO (graphs omitted, since they are similar to cases already discussed).

In practice, there is a management option of keeping the server utilization very low, e.g., the 18% reported by Gartner [44]. This is an expensive way to avoid long-tail response time problems.

Table 1. Summary of CTQO Observed

General case	Upstream CTQO	Downstream CTQO
Sync ⇒ Sync	Apache from Tomcat (Section 3.2) Apache and Tomcat from MySQL (Section 3.3) Tomcat from MySQL (Section 4.2)	Misaligned configuration settings (Section 4.5)
Async ⇒ Sync	No (Sections 4.3 and 4.4)	NginX ⇒ Tomcat (Section 4.2) XTomcat ⇒ MySQL (Section 4.3)
Async ⇒ Async	No (Sections 4.3 and 4.4)	No (Section 4.4)
Sync ⇒ Async	Apache from XTomcat (Section 4.5)	No (Section 4.5)

Our study shows that downstream CTQO may occur at low utilization levels if the configuration settings are misaligned for two servers connected by RPC-style communications and sufficiently high workload bursts happen. Consider a scale-out facility (either automated or manual) to keep the utilization low when the load increases in a data center. Consider the 1/1/1 configuration in our experiments and suppose a workload change (to heavy application server load) causes the scale-out facility to add two Tomcat servers, increasing the MaxSysQDepth(Tomcat) from 200 to 600. Since the database tier is not the bottleneck, MaxSysQDepth(MySQL) remains unchanged, for example, at 200. The resulting configuration could cause downstream CTQO when a sudden burst of requests is successfully handled by the 3 Tomcat servers, sending many requests to MySQL that may exceed MaxSysQDepth(MySQL) and lead to dropped packets.

Our experimental study of the 1-1-1 configuration is successful in exposing both upstream and downstream CTQO in all four combinations of synchronous with asynchronous servers (Section 4.6 shows a summary). The previous management option suggests that our study only opened the door to many interesting possibilities in the design, implementation, and operations of distributed applications in data centers, since the successful scale-out of one tier (Tomcat) may lead to millibottlenecks and CTQO in other tiers that have low utilization.

4.6 Summary of Evaluation

We summarize the results from the detailed evaluation from Section 4.2 to Section 4.5 in Table 1. In the left column, we classify the n-tier system components into four categories according to their communications style and handling of messages: sync→sync, async→sync, async→async, and sync→async. By sync, we mean a server with synchronous (blocking) message API, where each message is handled by a thread from request to response. In contrast, async servers have an event-based asynchronous message API, where messages are accepted and inserted into a lightweight message queue for further processing by other threads.

For each category, the middle column contains the examples of upstream CTQO if applicable, and the right column contains the examples of downstream CTQO if applicable. The table shows that async upstream servers can remove upstream CTQO, and async downstream servers can remove downstream CTQO. Consequently, an n-tier pipeline consisting entirely of async servers can avoid both upstream and downstream CTQO problems.

In Table 2, we summarize the causes for upstream and downstream CTQO found in our experiments. The upstream CTQO is started by a millibottleneck in the downstream server (ctd_0), which causes the queues in an upstream synchronous server (ctd_1) to exceed its MaxSysQDepth. The downstream CTQO is started by an upstream server releasing bursts of requests and causing a downstream synchronous server to exceed its MaxSysQDepth. Perhaps somewhat ironically, an asynchronous upstream server is more capable of processing a larger number of requests within

Table 2. Summary on Causes of CTQO

	Upstream CTQO	Downstream CTQO
Causes	Millibottlenecks in downstream server cause upstream sync server to drop packets when MaxSysQDepth overflows	Bursts of requests from upstream server cause downstream sync server to exceed its processing capacity + MaxSysQDepth overflow
Examples	Apache from Tomcat (Section 3.2) Apache and Tomcat from MySQL (Section 3.3) Tomcat from MySQL (Section 4.2) Apache from XTomcat (Section 4.5)	NginX ⇒ Tomcat (Section 4.2, during millibottleneck in Tomcat) XTomcat ⇒ MySQL (Section 4.3, after millibottleneck in XTomcat)

a short window, and thus more likely to cause downstream CTQO in a synchronous server (cases studied in Sections 4.2 and 4.3).

5 RELATED WORK

In latency sensitive web applications (e.g., 99th or even 99.9th percentile latency) [2, 3, 12, 22, 27, 29, 46, 60, 61], the long-tail response time problem has received considerable attention recently. The previous work can be divided into three major categories: (1) identification of sources of long-tail response time problem and their solution by improving resource allocation (scheduling), (2) solutions to long-tail response time without identifying explicitly the source, and (3) evaluation of asynchronous event-based systems, compared to synchronous versions.

In the first category (identify specific sources of long-tail response time, often with solutions targeted for those sources), representative examples of research include: Cake [53] (reactive feedback-control scheduler for different workloads), Chase et al. [30] (adaption lag of feedback controller for dynamic scaling of storage), Domino [28] (prioritize using the longest-wait-time-first (LWTF) policy), DeepDive [39] (short-term interference of co-located VMs, identifiable by hardware performance counters), Li et al. [29] (several sources in hardware, OS, and application level in web servers), Berger et al. [6] (dynamical reallocation of cache resource based on different latency-aware requests), PriorityMeister [63] (tail latency reduction through combining priority scheduling and multi-stage per-workload rate limit), Terei et al. [47] and Wang et al. [57] (Java garbage collection), Wang et al. [55] (control system lag in dynamic voltage and frequency scaling), and Bobtail [61] (co-scheduling VMs with CPU bound tasks as one root cause). Our study follows the same philosophy of identifying a class of problems and then evaluate solutions. At the same time, we differ from, and complement, the previous work in this category by focusing on distributed system phenomena (CTQO) and solutions (asynchronous n-tier system architecture).

In the second category (proposed solutions to long-tail response time problem without identifying explicitly the source), representative examples of research include Dean et al. [12] (use hedged requests and tied requests over replicated services to bypass the long-tail response time problem in Google's interactive applications), C3 [46] (apply adaptive replica selection scheme to address performance fluctuations across Cassandra distributed database servers), and Jalaparti et al. [20] (present a holistic framework, Kwiken, which considers the latency distribution in each stage, the cost of applying individual techniques and the workflow structure to minimize end-to-end latency). These approaches typically exploit server replicas to bypass the requests that take unexpected long time to respond, without identifying explicitly the source of the delay. Our work focuses on a class of long-tail response time problems that result from queue overflow and dropped requests. Although replicated servers can solve more problems than asynchronous servers, the latter has the potential to lower the costs in data centers without sacrificing the quality of service for latency-sensitive applications.

In the third category (evaluation of asynchronous systems), we found many studies of single web servers adopting event-based asynchronous architectures [10, 18, 26, 41, 51, 52, 62]. These research efforts focus mainly on reducing the multithreading overhead of thread-based design, instead of solving a distributed system problem. Second, streaming processing systems [48, 49] mainly adopt asynchronous messaging for inter-node communication, however, these systems are not interactive, e.g., call/response by nature as in web-facing systems, so their application domain is orthogonal to our research problem. In contrast, our study focuses on the advantages and disadvantages of synchronous versus asynchronous communications in distributed n-tier systems.

6 CONCLUSION

Despite the progress made in recent years on the various methods to mitigate long-tail response time problems in web-facing applications [2, 3, 12, 22, 27, 29, 46, 61], they remain a serious threat that may be contributing to the continuing low utilization of servers in data centers [27, 32, 44]. These previous research efforts address mainly two classes of long-tail response time problems caused by either uneven workloads (some requests are intrinsic heavy) such as web search, or resource contention in single nodes. In this article, we focus on a third class that arises from Cross-tier Queue Overflow (CTQO), a distributed system phenomenon resulting from the RPC-style synchronous inter-node communication.

Our study shows that CTQO happens in a classic n-tier configuration (Apache, Tomcat, and MySQL) with the following causal chain of events: (1) occurrence of millibottlenecks with tens to hundreds of duration at moderate average utilization, (2) CTQO causing a synchronous server to exceed its MaxSysQDepth (worker thread pool size plus the TCP buffer size), (3) excess packets are dropped, (4) retransmission of dropped packets, (5) long-tail response time due to response times of multiple seconds for the retransmitted packets. CTQO is a broad problem, since the initiating millibottleneck can arise from resources in any system layer (e.g., CPU, memory, network and disk I/O), as shown by previous work [56, 57]. To address the CTQO challenge, we replace the synchronous servers (Apache, Tomcat, MySQL) one-by-one with their asynchronous counterparts: Nginx, and asynchronous versions of Tomcat and MySQL. The experiments evaluated in detail all viable combinations between synchronous and asynchronous servers.

We found two main CTQO scenarios. The first one is *upstream* CTQO when dropping requests occurs in an upstream server due to the millibottlenecks in a downstream server pushing more requests to queue upstream. The second one is *downstream* CTQO when dropping requests occurs in a downstream server, because the millibottlenecks in its upstream server make accumulated queued requests suddenly flood to downstream. Once we replace all thread-based synchronous servers with their asynchronous counterparts, both upstream & downstream CTQO can be avoided even if the system runs at moderate utilization levels. Our research suggests that by applying asynchronous inter-tier communications for the entire n-tier system, we may effectively reduce the non-trivial long-tail response time problems resulting from CTQO.

APPENDICES

A EXPERIMENTAL SETUP

We use a standard n-tier benchmark RUBBoS as a representative interactive online applications in our experiments. RUBBoS benchmark application is modeled after the popular tech news website Slashdot [1]. A typical configuration for the RUBBoS benchmark application is a three-tier architecture (a Web server tier, an application server tier, and a database tier). More tiers (e.g., Cache, load balancer) can be added based on the application need. Each tier communicates with each other using the classic RPC-style synchronous request-response. The benchmark application supports

Software Stack

PRC Web Server	Apache 2.2.22 + tomcat-connectors-1.2.28
Asyn Web Server	Nginx-1.6.2
RPC App Server	Tomcat 7.0.57 + BIO connector + mysql-connector-java-5.1.19
Asyn App Server	Tomcat 7.0.57 + NIO connector + async-mysql-connector-1.0
Database server	MySQL 5.5.19
Operating system	RHEL 6.3 (kernel 2.6.32)
Hypervisor	VMware ESXi v5.0

(a) Software setup

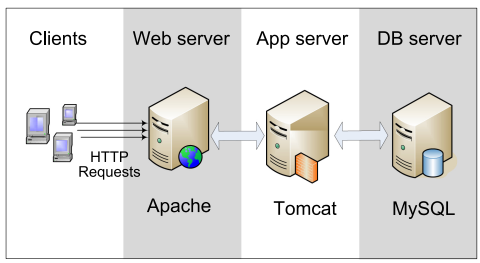
ESXi Host Configuration

Model	Dell Power Edge T410
CPU	2* Intel Xeon E5607, 2.26GHz Quad-Core
Memory	16GB
Storage	7200rpm SATA local disk

VM Configuration

Type	# vCPU	CPU limit	CPU shares	vRAM	vDisk
Small (S)	1	2.26GHz	Normal	2GB	20GB

(b) ESXi host and VM setup



(c) 1/1/1 sample topology

Fig. 15. Experimental setup.

24 different web interactions such as ViewStory and StoriesOfTheday. The workload generator of this benchmark supports two workload modes: browse-only CPU intensive and read/write mixes. We use the former mode in this article. The workload generator launches a certain number of threads, each of which simulates the behavior of a normal user when interacting with the benchmark application. Thus, the workload intensity can be controlled by specifying the number of threads in the workload generator. Such as workload generator design is similar to that of many other n-tier benchmarks such as RUBiS and Cloudstone.

We conduct our experiments on our virtualized cluster environment. Figure 15 illustrates the hardware and software configurations, and a simple three-tier configuration adopted in our experiments. Every server is hosted by one virtual machine (VM). Each VM is deployed on a dedicated physical machine unless explicitly specified to conduct VM co-location experiments.

B CONNECTORS FOR ASYNCHRONOUS INTER-TIER COMMUNICATION

A key unit for inter-tier communication in an n-tier system is the *connector*. Each server uses a connector to communicate with other servers in the system (see Figure 8). The main activities of a connector are to manage incoming and outgoing network connections, parse and route the incoming requests to the application layer (business logic), and write response back to clients through established connections. Both the synchronous and the asynchronous connectors share similar functionality in high level, but they have very different mechanisms to interact with the OS and the application layers.

Thread-based servers mainly use synchronous connectors with RPC-style request-response for inter-node communication. When a synchronous connector accepts a request, it will dispatch the request to a dedicated worker thread for handling until the finish of the request. Thus, each concurrent request consumes one worker thread of the server. The concurrent request processing is realized by the operating system transparently switching among worker threads (thus context switch occurs). Although widely used in production internet servers, synchronous connectors bring two

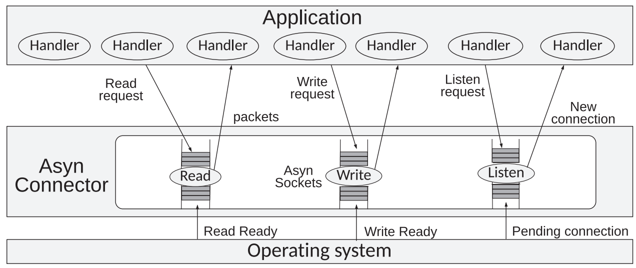


Fig. 16. Interaction between an asynchronous connector with the application and the OS layers.

problems when handling high concurrent requests. The first one is the multithreading overhead, which has been extensively studied before [9, 26, 41, 54, 58, 59]. The second but more interesting one is the Cross-tier Queue Overflow as discussed in this article.

An asynchronous connector follows the event-driven design. Previous research efforts have shown that the asynchronous event-driven architecture could be a superior alternative to the traditional thread-based design by mitigating the multithreading overhead [11, 26, 41, 59, 62]. However, it is challenging to build high-performance asynchronous event-driven servers due to the obscured non-sequential control flow rooted in the event-driven programming model. An asynchronous connector manages a bunch of connections and interacts with both the application and the OS layer through one or a few threads handling various events (see Figure 16). Concretely, an asynchronous connector handles events received from both the application and the OS layers by looping over two phases. The first phase is responsible for events monitoring, determining which connections have pending events (read or write) that need to be processed. The asynchronous connector is able to achieve this by exploiting the event notification mechanisms supported by the underlying operating system (e.g., *epoll* for Linux).

The second phase is responsible for event processing. In this phase, a scheduling thread pulls out those connections with pending events and iterates over each connection. During the iterating process, the scheduling thread calls the appropriate event handler (based on the context information) to handle each event. We note that theoretically only one thread is needed to loop over the two phases. In reality, multiple threads can be allocated to each phase in case of transient disk I/O blocking or efficiently utilizing a multi-core CPU [41].

The asynchronous connector design suggests the decoupling of component servers in the request processing chain of an n-tier system. One or a few processing threads of each server loops continuously over the two phases in each asynchronous connector, processing various types of local events, and are independent of the queuing status of servers in other tiers. In this case, the queue of a downstream server will not be pushed back to the upstream tiers, thus break the channel of Cross-tier Queue Overflow.

In our experiments, we directly use or implement a few asynchronous servers, shown in Figure 8. For example, we use a popular asynchronous web server Nginx [38]. The asynchronous XTomcat [4] is based on the Tomcat version 7 (the latest version at the time), which supports an asynchronous connector to handle upstream communication. We modified an open source asynchronous JDBC driver [16] for XTomcat to support asynchronous invocation with the downstream database.

For the database tier, XMySQL simulates an asynchronous MySQL by adopting the InnoDB storage engine of MySQL, which supports a lightweight queue to store the waiting queries. Specifically, the InnoDB allows MySQL to limit the number of active threads for query processing; additional queries that exceed the thread limit will be stored into a FIFO queue to avoid high concurrency overhead. In our experiments, we set the active thread limit in MySQL to be 8 while setting the limit of accepted queries to a large number (2,000) to avoid dropped queries.

C ASYNCHRONOUS BENCHMARK APPLICATION CONVERSION

Most existing n-tier benchmark applications (e.g., RUBiS, Cloudstone, TCP-W) are implemented using the traditional synchronous programming model. With this programming model, each accepted request is processed in a straightforward sequential fashion by a dedicated server thread. Specifically, the application level business logic use synchronous RPC request-response to communicate with other servers; the processing thread will block until each synchronous call returns (e.g., returned ResultSet from a database query). Thus even if a server adopts asynchronous connectors, the original benchmark application with sequential control flow is not compatible with the asynchronous I/O abstractions (i.e., read, write, and listen) provided by the asynchronous connectors. To make an asynchronous benchmark application, the original synchronous benchmark application needs to be re-implemented using the event-driven programming model and make it use the asynchronous connectors to conduct inter-tier communication.

In the event-driven programming model, the processing of each request is divided into multiple disjoint stages, the execution of each stage is triggered by an event. Figure 17 illustrates the process of converting a simple RPC-style synchronous Java servlet to its event-driven version. The logic of the synchronous Java servlet is an abstraction of the sequential execution of a set of ordinary synchronous database queries, *SyncDBQuery*1, m, *SyncDBQueryN*, inside a Tomcat application server. The servlet will process the returned result of each synchronous database query before moving the next synchronous database query. Such a simple servlet can be systematically transformed into the functionally equivalent asynchronous version as shown in the right part of Figure 17. This figure shows that once we see a synchronous database query (e.g., *SyncDBQuery*1, *SyncDBQuery*2), the original sequential logic needs to split into two functions. The first function will execute the original database query in a non-blocking mode; the second function is referred as the call-back function, which will be triggered only when the previous database query returns results (thus network I/O events).

We note that Figure 17 is just a simple example of converting the basic sequential flow to its asynchronous version. Schneider [50] introduced some transformation rules that help convert more complicated control flows such as for-loop and switch statement from the synchronous version into their asynchronous version. In addition, these conversion rules are not only applicable to stateless programs, but they can also be applied to stateful programs given that a global context object associated with each client request can be passed to an asynchronous call. Using Schneider's

```
function doGet(request1) {
    ... pre-processing request1 ...
    ... form query1 ...

    result1=SyncDBQuery1(query1);
    ... think about result1 ...
    ... form query2 ...

    result2=SyncDBQuery2(query2);
    ... post-processing result2 ...
    ... form response ...

    return response;
}
```

(a) A simple synchronous Java Servlet

```
function doGet(request1) {
     ... pre-processing request1 ...
     ... form query1 ...
     AsynDBQuery1(query1, eventHandler1);
}

function eventHandler1(result1) {
    ...think about result1 ...
    ...form query2 ...
    AsynDBQuery2(query2, eventHandler2);
}

function eventHandler2(result2) {
     ...post-processing result2...
     ...form response ...
     return response;
}
```

(b) An asynchronous event-driven Java servlet in XTomcat

Fig. 17. Simple RPC transformed into a set of asynchronous calls.

transformation rules, we have successfully transformed the RUBBoS benchmark application into its asynchronous version, which enables the large-scale performance evaluation of the asynchronous architecture of n-tier applications.

REFERENCES

[1] Stephen Adler. 1999. The Slashdot effect: An analysis of three Internet publications. *Linux Gazette* 38 (1999), 2.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*. 63–74.

[3] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. 253–266.

[4] Apache Software Foundation. 2019. Java Non Blocking Connector (NIO). Retrieved from https://tomcat.apache.org/tomcat-7.0-doc/config/http.html.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 164–177.

[6] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 195–212.

[7] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst.* 2, 1 (Feb. 1984), 39–59. DOI: https://doi.org/10.1145/2080.357392

[8] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2010. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 241–252.

[9] Hui Chen, Qingyang Wang, Balaji Palanisamy, and Pengcheng Xiong. 2017. DCM: Dynamic concurrency management for scaling n-tier applications in cloud. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. IEEE, 2097–2104.

[10] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazires, and Robert Morris. 2002. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*. 186–189.

[11] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node. fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the T12th European Conference on Computer Systems*. ACM, 145–160.

[12] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[13] Christina Delimitrou and Christos Kozyrakis. 2018. Amdahl's law for tail latency. *Commun. ACM* 61, 8 (2018), 65–72.

[14] Qi Fan and Qingyang Wang. 2015. Performance comparison of web servers with different architectures: A case study using high concurrency workload. In *Proceedings of the 3rd IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb'15)*. IEEE.

[15] Jim Gettys and Kathleen Nichols. 2012. Bufferbloat: Dark buffers in the internet. *Commun. ACM* 55, 1 (2012), 57–65.

[16] Google Code Archive. 2009. Non-Blocking (asynchronous) MySQL Connector for Java. Retrieved from https://code.google.com/archive/p/async-mysql-connector/.

[17] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*. 22.

[18] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. 2012. Comparing high-performance multi-core web-server architectures. In *Proceedings of the 5th Annual International Systems and Storage Conference*. 1.

[19] Instagram Engineering. 2018. Open-sourcing a 10x reduction in Apache Cassandra tail latency. Retrieved from https://instagram-engineering.com/open-sourcing-a-10x-reduction-in-apache-cassandra-tail-latency-d64f86b43589.

[20] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, vol. 43. ACM, 219–230.

[21] Deepal Jayasinghe, Calton Pu, Tamar Eilam, Malgorzata Steinder, Ian Whally, and Ed Snible. 2011. Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *Proceedings of the IEEE International Conference on Services Computing (SCC'11)*. IEEE, 72–79.

[22] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. 2016. TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 129–141.

[23] Yasuhiko Kanemasa, Qingyang Wang, Jack Li, Masazumi Matsubara, and Calton Pu. 2013. Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation. In *Proceedings of the 10th IEEE International Conference on Services Computing (SCC'13)*. 136–143.

[24] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. 9:1–9:14.

[25] Ron Kohavi and Roger Longbotham. 2007. Online experiments: Lessons learned. *Computer* 40, 9 (2007), 103–105.

[26] Maxwell N. Krohn, Eddie Kohler, and M. Frans Kaashoek. 2007. Events can make sense. In *Proceedings of the USENIX Annual Technical Conference*. 87–100.

[27] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. 4:1–4:14.

[28] Ding Li, James Mickens, Suman Nath, and Lenin Ravindranath. 2015. Domino: Understanding wide-area, asynchronous event causality in web applications. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC'15)*. ACM, New York, NY, 182–188. DOI: https://doi.org/10.1145/2806777.2806940

[29] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*. New York, NY.

[30] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. 2010. Automated control for elastic storage. In *Proceedings of the IEEE International Conference on Autonomic Computing (ICAC'10)*.

[31] LinkedIn Engineering. 2015. Who moved my 99th percentile latency. Retrieved from https://engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency.

[32] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2016. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.* 34 (2016), 6:1–6:33. Retrieved from http://dl.acm.org/citation.cfm?id=2882783.

[33] Simon Malkowski, Yasuhiko Kanemasa, Hanwei Chen, Masao Yamamoto, Qingyang Wang, Deepal Jayasinghe, Calton Pu, and Motoyuki Kawaba. 2012. Challenges and opportunities in consolidation at high resource utilization: Non-monotonic response time variations in n-tier applications. In *Proceedings of the IEEE 5th International Conference on Cloud Computing (CLOUD'12)*. IEEE, 162–169.

[34] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. 2008. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference (Middleware'08)*. 265–286.

[35] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. 2009. Injecting realistic burstiness to a traditional client-server benchmark. In *Proceedings of the 6th International Conference on Autonomic computing (ICAC'09)*. 149–158.

[36] Jeffrey C. Mogul. 2006. Emergent (mis) behavior vs. complex software systems. *ACM SIGOPS Operat. Syst. Rev.* 40, 4 (2006), 293–304.

[37] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, 237–250.

[38] NGINX. 2017. nginx. Retrieved from http://nginx.org/.

[39] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Annual Technical Conference*. 219–230.

[40] ObjectWeb Consortium. 2005. RUBBoS: Bulletin board benchmark. Retrieved from http://jmob.ow2.org/rubbos.html.

[41] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. 2007. Comparing the performance of web server architectures. In *ACM SIGOPS Operating Systems Review*, vol. 41. 231–243.

[42] Junhee Park, Qingyang Wang, Jack Li, Chien-An Lai, Tao Zhu, and Calton Pu. 2016. Performance interference of memory thrashing in virtualized cloud environments: A study of consolidated n-tier applications. In *Proceedings of the IEEE 9th International Conference on Cloud Computing (CLOUD'16)*. IEEE, 276–283.

[43] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. 2011. *Computing TCP's Retransmission Timer*. Technical Report.

[44] Bill Snyder. 2010. Server virtualization has stalled, despite the hype. Retrieved from https://www.infoworld.com/article/2624771/server-virtualization-has-stalled--despite-the-hype.html.

[45] SOURCEFORGE. 2018. Collectl. Retrieved from http://collectl.sourceforge.net/.

[46] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. 513–527. Retrieved from http://dl.acm.org/citation.cfm?id=2789770.2789806.

[47] David Terei and Amit Levy. 2015. Blade: A data center garbage collector. *arXiv preprint arXiv:1504.02578.*

[48] The Apache Software Foundation. 2018. Apache Flink. Retrieved from https://flink.apache.org/.
[49] The Apache Software Foundation. 2018. Apache Storm. Retrieved from http://storm.apache.org.
[50] Thibaud Lopez Schneider. 2008. Writing Effective Asynchronous XmlHttpRequests. Retrieved from https://www.thibaudlopez.net/xhr/Writing%20effective%20asynchronous%20XmlHttpRequests.pdf.
[51] Robert von Behren, Jeremy Condit, and Eric Brewer. 2003. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS'03)*. 19–24.
[52] Rob Von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: Scalable threads for internet services. In *ACM SIGOPS Operating Systems Review*, vol. 37. 268–281.
[53] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. 2012. Cake: Enabling high-level SLOs on shared storage systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2391229.2391243
[54] Qingyang Wang, Hui Chen, Shungeng Zhang, Liting Hu, and Balaji Palanisamy. 2019. Integrating concurrency control in n-tier application scaling management in the cloud. *IEEE Trans. Parallel Distrib. Syst.* 30, 4 (2019), 855–869.
[55] Qingyang Wang, Yasuhiko Kanemasa, Chien-An Li, Jack Lai, Masazumi Matsubara, and Calton Pu. 2013. Impact of DVFS on n-tier application performance. In *Proceedings of ACM Conference on Timely Results in Operating Systems (TRIOS'13)*. 33–42.
[56] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, and Calton Pu. 2013. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS'13)*. 31–40.
[57] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien-An Lai, Chien-An Cho, Yuji Nomura, and Calton Pu. 2014. Lightning in the cloud: A study of very short bottlenecks on n-tier web application performance. In *Proceedings of USENIX Conference on Timely Results in Operating Systems (TRIOS'14)*.
[58] Qingyang Wang, Simon Malkowski, Yasuhiko Kanemasa, Deepal Jayasinghe, Pengcheng Xiong, Calton Pu, Motoyuki Kawaba, and Lilian Harada. 2011. The impact of soft resource allocation on n-tier application scalability. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. 1034–1045.
[59] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. 230–243. DOI: https://doi.org/10.1145/502034.502057
[60] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. 2013. Small is better: Avoiding latency traps in virtualized data centers. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*.
[61] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. 2013. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. 329–342.
[62] Shungeng Zhang, Qingyang Wang, and Yasuhiko Kanemas. 2018. Improving asynchronous invocation performance in client-server systems. In *Proceedings of the IEEE 38th International Conference on Distributed Computing Systems (ICDCS'18)*. IEEE, 907–917.
[63] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2014. PriorityMeister: Tail latency QoS for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2670979.2671008

Received February 2019; revised April 2019; accepted June 2019